A Fault-Tolerant Session Layer with Reliable One-Way Messaging and Server Migration Facility

Naghmeh Ivaki, Serhiy Boychenko, Filipe Araujo
CISUC, Dept. of Informatics Engineering, University of Coimbra, Portugal
naghmeh@dei.uc.pt, serhiy@student.dei.uc.pt, filipius@uc.pt

***Abstract*—Despite being extremely successful, TCP has a number of shortcomings when network disruptions occur, or when peers do not follow a request-reply interaction: it does not handle connection crashes, event-driven communication or application migration. In many cases, programmers must engineer their own solutions to write reliable distributed applications.**

To overcome these limitations, we propose FTSL, a Fault-Tolerant Session Layer that works on top of TCP. Besides offering a full-duplex connection, FTSL owns a number of distinctive features: it tolerates TCP connection crashes, it provides highly decoupled reliable patterns for one-way communication, and it enables server-side migration. While the first two greatly simplify distributed systems programming for a wide range of applications, the latter enables cloud systems managers to move a server application for load balance or maintenance, without moving the entire virtual machine.

We present the FTSL protocol, its implementation, and resort to performance to show that FTSL imposes a reasonable overhead for the guarantees it provides.

***Index Terms*—Fault Tolerance, Application Migration, One-Way Messaging, TCP, Session Layer**

I. Introduction

Reliable communication lies at the heart of distributed systems. Over the last few decades, the Transmission Control Protocol (TCP) [11] has been the most common option to emulate reliable communication over the Internet. Unfortunately, TCP has several limitations. In this paper we address three of them. First of all TCP does not address connection failures. If connectivity fails for some time, endpoints may receive an exception and fail to determine which messages did or did not reach the peer, even in request-reply interactions, thus being unable to rollback to some coherent state. However, for many operations, such as file transfers, secure shell interaction, web streaming or messaging, surviving TCP disconnections might be very practical. In fact, we dare to say that the TCP/IP stack lacks a session layer capable of making through disconnections.

TCP provides a full-duplex communication channel, well suited for a large range of client-server applications. But not all patterns fit into this spectrum. Sending a message to a peer without expecting any reply is the simplest possible coordination pattern. One-way messaging is extremely useful in event-based systems, where information flows in a single direction. Group communication, broadcasting mechanisms, publish-subscribe, or complex event processing systems are some of the scenarios that need this messaging pattern. We can find it in some forms of remote procedure calls, including web services. In some cases, the web client may not need a reply (although it wants the server to handle the request), but they are bound to communicate in a request-response model. We could think for example of online multi-player games. TCP provides no special coordination for this type of peers, and reliability is simply not enough, because the sender cannot tell whether the receiver processed data correctly. Applications could implement confirmations over TCP (e.g. using a request-response model), but besides being evidently not one-way communication, the need to wait for each single response would undoubtedly slow down performance.

We also address a third limitation of TCP. Especially in the era of cloud computing and virtualization, it would be quite valuable to move server from one machine to another without disrupting services. In fact, migrating applications or virtual machines inside the same cloud or even among different clouds is a very useful feature that deserved already some attention in the existing literature [22], [3], [30], [14]. Unfortunately, migrating applications is very hard to accomplish with TCP, because connections would all crash, and the IP address of the receiving machine would be different. Even a virtual machine migration might be troublesome if it takes too long. By addressing all the above issues, we can make distributed applications, including those running in cloud environments, more dependable.

To overcome the limitations of TCP, we propose a Fault Tolerant Session Layer (FTSL). FTSL is a message-oriented reliable session protocol that provides special support for one-way communication, besides enabling live server migration from machine to machine, regardless of their IP addresses. We compare FTSL with a number of related solutions. FTSL compares favorably in terms of performance with the heaviest solutions, like Java Message Service [25], and shows a reasonable degradation against lighter solutions such as TCP or an optimized version of RSocket [23]. We present the challenges, the design and some points of future improvement.

This paper is organized as follows. The next section provides a review of related work. Section 3 describes FTSL: its reliability, one-way messaging and server migration. Section 4 presents FTSL's architecture and implementation. Section 5 presents the experimental evaluation. Section 6 concludes this paper.

II. Related Work

TCP uses sequencing, acknowledgments and retransmission to detect and repair packet losses, but it cannot overcome longer communication outages. Normally, the maximum tolerable disconnection period is between 30 and 90 seconds [1]. Recovery from the outages is more complex than it looks like, because programmers cannot easily determine which messages did or did not get through. We found a large body of work in the literature to solve disconnection and crash problems.

SCTP [21] provides support for tolerating network congestion and failure by establishing multiple redundant paths between the client and the server. Like in our own system, applications may not use TCP directly, but the tolerance of SCTP comes from the redundancy, alone. Moreover, SCTP does not focus on one-way messaging or relocation of the server.

Our system also bears some similarities to solutions that insert a layer between TCP and the application. This layer reestablishes the lost sessions after reconnecting [1]. Another work targeting some (but not all) of the FTSL goals is Robust Socket (RSocket) [23], which solves the problem of broken TCP connections. RSocket changes some Java core libraries, leaving the standard Java TCP interface untouched. To recover from TCP crashes it uses control messages over a separate channel.

Zandy and Miller propose an even simpler idea in [29]. Authors use an in-flight circular buffer to store sent messages. The size of this buffer is the sum of the size of the local TCP send buffer and the size of the remote TCP receive buffer. When the application sends a message, a copy is saved into the in-flight buffer, and a sent bytes counter is updated. Since it is circular, when the buffer fills, new data automatically delete the oldest. Receivers also store the number of bytes read by the application. Once peers recover from a connection failure, they exchange this counter, to let each other know what to resend. With this mechanism there is no need for extra acknowledgements during an interaction.

Other systems redirect all TCP traffic between client and server through a proxy that keeps the state of the connection. If the server crashes, the proxy switches the connection to a backup server and ensures that the new connection is consistent with the client's state. This approach does not tolerate TCP connection failures caused by network crashes. Furthermore, the proxy is a new single point of failure [17]. Similarly, HydraNet-FT [28] replicates services across an internetwork and provides a single, fault-tolerant service. It uses TCP with a few modifications on the server side. We can also find related solutions in ST-TCP [18], an extension of TCP to overcome server failures. In MI_TCP [13] servers of a cluster write a checkpoint of the TCP connection state. This allows the TCP connection to migrate to another server in the cluster and enables servers to transparently take over the connection. ER-TCP [27] combines primary-backup replication with logging, to achieve fault-tolerance on the server side of TCP connections. Orgiyan and Fetzer in [20] propose a TCP tapping approach to replication that masks TCP endpoint failures, such as server host crashes and server process crashes. Many of these systems tolerate server crashes, but not network crashes. Some of these solutions also require kernel modifications.

Another purpose of FTSL is to reconcile the conflicting goals of one-way messaging and reliable delivery of messages. In the (blocking) request-reply paradigm, we can find technologies supporting different levels of reliability. For example, Remote Procedure Calls (RPCs) or variants with at-least-once [4] or at-most-once [8] message delivery. Web services [6] provide an interesting example, because they are usually request-response, but may also support one-way messaging. However, the sender will not know anything about the result of its operations. Many messaging systems provide truly one-way reliability to clients, e.g., Microsoft's MSMQ [10] or the Java Message Service (JMS) [25], supported by a very large number of messaging stacks, like Websphere MQ [9], JBoss [12], HornetQ [16], etc. The message broker may ensure guaranteed delivery, and strong decoupling between peers, but it also makes communication slower. In some technologies, the broker may also be a single point of failure. Messaging systems are excellent for publish-subscribe systems or for reliable (even transactional) point-to-point communication. Their shortcoming is the overhead, the complexity of the infrastructure and of the programming APIs.

Checkpointing and restarting applications to load balance or to tolerate faults is not a new idea. Checkpointing is particularly useful for grid or cluster computing [15], [2], [19], where faults or shutdowns may wipe many hours of computation. Checkpointing may occur at the user or kernel level. The former requires fewer configuration, but fails to properly save kernel state. Checkpointing distributed applications is a challenge of its own. Solutions for this exist, but they are complex [7]. We address the simpler case of server checkpointing, which is very useful for the extremely frequent client-server paradigm and for cloud services that one may need to migrate. This approach is much lighter than migrating entire Virtual Machines.

III. The Fault-Tolerant Session Layer

FTSL is a TCP-based reliable, connection and message-oriented session layer protocol. It offers applications the possibility of keeping interactions by surviving network failures. Unlike TCP, which is stream-oriented, in FTSL applications explicitly mark message boundaries. For performance reasons, applications may send their messages in multiple chunks of data. By relying on the TCP/IP layers, who are often limited by NAT or firewall schemes, the initiative to connect or to recover from network failures always belongs to the client side.

A. Protocol Operation Overview

We use a *Sent Buffer* to keep sent, but unconfirmed FTSL packets and a *Received Buffer* to keep unread messages. The size of the sent buffer is limited to the the size of its equivalent in the TCP socket, to avoid blocking write operation in FTSL, due to lack of space. Unlike the approaches we reviewed in Section II, e.g. [29], we could not recover from TCP connection crashes with one single buffer on the sender side, because FTSL does more than simple connection recovery. For example, FTSL enables applications to confirm the reception of messages, thus requiring some out-of-band data, like acknowledgements, that must not wait in the buffers.

Since the same channel is used for both data and control messages, in contrast to RSocket, FTSL needs to keep reading from the channel to receive control messages. A receive buffer is therefore necessary as well. It allows the FTSL to read from the TCP socket in parallel. We ruled out the possibility of using urgent TCP data, because the TCP stacks seem to suffer from multiple issues, including incompatible interpretations of RFC 793 [11] and incorrect implementations of RFC 1122 [5].

All messages sent by the application go through FTSL, which adds its own header. This header includes the session ID (SID), which uniquely identifies each session, a flag to identify the type of the message, a packet ID (PID), and a message ID (MID). These uniquely identify each packet and each message. FTSL piggybacks the acknowledgment of the last packet received (rPID), the last message delivered to the application (dMID), and the last message delivered and processed by the application (pMID) in the header.

The server side generates a unique SID that lasts for the life of the session. We use a flag field to identify the type of the message such as APP, ACK, and CLS. APP serves for data messages from the application layer. FTSL explicitly marks the last packet of a message. ACK and CLS are control messages. The peers send acknowledgments (ACK) periodically if some messages remain unacknowledged. CLS is sent to notify the other party that the connection is going to be closed. FTSL closes the connection only if the application explicitly requests this.

B. End-to-End Reliability For One-Way Messaging Pattern

We argue that the right solution for reliable one-way messaging pattern is somewhere in-between the extremes of never sending any feedback and the request-reply paradigm. On one hand, closing the loop and letting the sender know the result of its actions enables the creation of more reliable applications. On the other hand, we must not do it on a single-message basis, because this is too costly. The mechanism we propose in FTSL offers end-to-end reliability to one-way operations, by using an asynchronous confirmation mechanism. This unties the sender from the receiver to let them do independent progress. Refer to Figure 1. The sender uses FTSL to send its message and gets back an identifier from the session layer, which saves and sends the message using TCP. To support a large number of receivers, the sender proceeds without blocking after sending the message. However, it may later know what happened to the message, depending on the options it picks for that specific message at send time. It can get a session-layer acknowledgment, or an application-layer acknowledgment. In the former case, the receiver's session layer acknowledges the message at reception time (in fact for performance reasons we usually delay this operation). In the latter case, the receiver application acknowledges the message after processing it, either explicitly or automatically. In the explicit confirmation the application calls an FTSL method. The automatic confirmation occurs once a listener method of the receiver invoked by FTSL finishes. To do the FTSL-level confirmation we rely on the rPID and dMID, while for the application-level confirmation we rely on pMID.

The error case is different. For some reason, the receiver may be unable to acknowledge the message, e.g., because its contents trigger some exception. If the receiver does not acknowledge and calls `recv()` again, the FTSL will redeliver the same message, using a special flag to inform the receiver of that fact. After a predetermined number of failures, in the next `recv()`, FTSL delivers the next message and sends back an indication of error to the sender side. This bears some similarities to the `DUPS_OK_ACKNOWLEDGE` mode of JMS, where clients can get repeated messages. In both cases, of success or error, once the acknowledgment gets to the sender side, the application can get it using an explicit call or through a previously registered callback method. We think that this does not violate the logic of one-way communication, because neither of these methods forces the sender to block at send time.

C. Checkpointing for Server Migration

FTSL lets server applications move from one machine (or port) to another, something that is particularly useful in cloud environments. This releases managers from the burden of halting and moving entire virtual machines, for maintenance or load balancing. We limit migration operations to the endpoints that *only accept* connections, which is the case of typical servers. The idea is that a server usually does not start connections to other endpoints, or else we would need a distributed checkpoint algorithm, such as the ones of [7], [26]. Despite being simpler, we believe that this alternative covers many (if not most) cases of interest, because the client-server paradigm is ubiquitous.

To enable the migration mechanism we need a naming service that takes the name of an FTSL session and gives back the IP address and port of the server. We use an RMI object for that purpose and a simple rebind and unbind pair of operations. The only thing that clients need to know is the (fixed) address of the naming server, to effectively allow the FTSL server to change place. We did not use the Java Naming and Directory Interface (JNDI) shipped with GlassFish for this task, because we did not manage to make it run efficiently. Sometimes it could be 5 to 10 times slower than what we eventually managed to do.

At the heart of this process is the Java serialization mechanism, which enables the server to save the Java classes implementing FTSL. Once FTSL receives an order to serialize, it must halt its actions and it must not handle further packets from the client to preclude any state that is *subsequent* to the checkpoint. FTSL will not resume before receiving another explicit request from the application. The serialization order completely freezes the protocol on the server side. From the point of view of the client, once TCP signals the end of the connection, FTSL starts the reconnection procedure, which involves a naming server lookup followed by a TCP connection. Since the client keeps its internal FTSL state (buffers, variables, etc.) and so does the server, the next time they restart, the buffers are in the same state, the difference being the channel that has no messages among them. This may prompt retransmissions, if acknowledgments are missing.

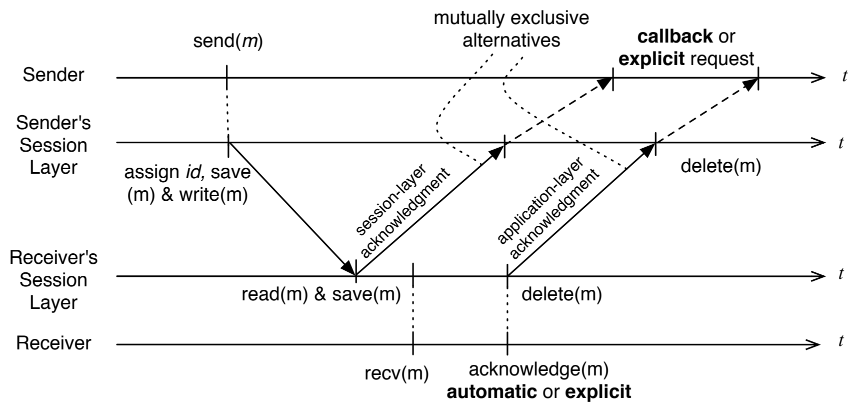
send(m)
mutually exclusive alternatives
callback or explicit request
Sender
Sender's Session Layer
assign id, save (m) & write(m)
session-layer acknowledgment
application-layer acknowledgment
delete(m)
Receiver's Session Layer
read(m) & save(m)
delete(m)
Receiver
recv(m)
acknowledge(m) automatic or explicit
t

Fig. 1. Successful one-way messaging pattern

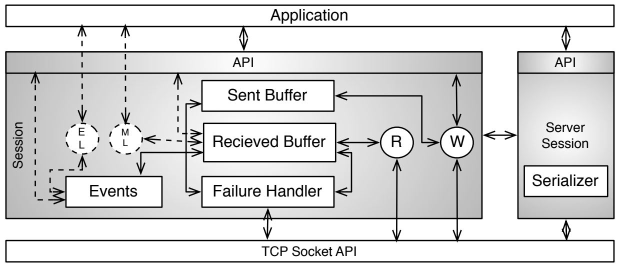

Fig. 2. FTSL Architecture

IV. FTSL Implementation and API

A. Implementation

Figure 2 displays the most important components of FTSL. We implemented FTSL in Java. The classes `ServerSession` and `Session` are inspired on their TCP counterparts, the `ServerSocket` and `Socket`. An instance of `ServerSession` class is created by the server to wait for session requests from the clients. Once a connection is setup, the server gets an instance of the class `Session`, which is connected to a similar one on the client. Most of the actual functionalities of FTSL are implemented in this `Session` class. It provides send and receive operations to the peers and takes the responsibility of keeping track of the message flow. Upon creation of the session, an instance of `ReceivedBuffer`, `SentBuffer`, and `Event` are created inside the session. The messages sent by the application are saved in the `SentBuffer` and are deleted after acknowledged. The received messages are saved in the `ReceivedBuffer` and are deleted either after being delivered to the application layer (for the messages that do not need the confirmation) or after being confirmed by the application (otherwise). To control concurrency coming from simultaneous access of FTSL and application threads to the `SentBuffer` and `ReceivedBuffer` objects, we used a `java.util.concurrent BlockingQueue`. The `Failure Handler` serves to restore the session whenever the TCP connection crashes. This involves setting up a new TCP connection, followed by the process of resending all messages unacknowledged by the peer.

To receive messages, the application may implement the `MessageListener` interface or may call API methods explicitly. In the former case, delivery of a message will occur immediately as the `Session` object calls back the appropriate user method. FTSL also uses events to notify applications of changes in the status of the previously sent messages. These are `Event` objects that applications may, likewise, receive synchronously through API method calls or asynchronously through previously registered `EventListener` callbacks.

Blocking send operations are taken care by the application main thread, whereas non-blocking ones are controlled by a session thread. In the `Session` object, a new thread is also created to read the messages from the input stream. Depending on the use of listeners, one or two more threads might be created in the `Session`: one to callback the application for messages, another one for events. There is still one more thread in the session taking care of the acknowledgements. Although, an acknowledgment should be sent for each message, to decrease the overhead of the system we often wait to piggyback the same data in the send messages. Since this could result in long periods without sending any acknowledgment, we dedicate an extra thread to periodically check if there is any unacknowledged message.

We implemented the Java `Serializable` interface in the `ServerSession` and `Session` classes to enable checkpointing the state of active server-side FTSL sessions. We intercept and serialize all the sessions using `Serializer` to enable the server to move all active sessions to a new machine without losing the state of the interactions.

B. The Application Programming Interface of FTSL

We show the main part of the FTSL's API in Table I. It provides four major operations to the applications regarding its core functionalities: creation of a session, termination of a session, sending to and receiving from the session. The server creates a `ServerSession` and waits in the `accept` method for a session connection request, whereas the client takes the initiative to create a session, by instantiating a new `Session` object. The server can intercept and save all the active sessions

TABLE I
FTSL API

ServerSession	**Exceptions**
ServerSession(address, port)	IOException
Session ServerSession.accept()	-
void ServerSession.close()	-

Session	**Exceptions**
Session(address, port)	UnknownHostException,IOException
int getSessionID()	-
InetAddress getInetAddress()	-
int getPort()	-
InetAddress getLocalAddress()	-
int getLocalPort()	-
void setEventListener(EventListener el)	-
void setMessageListener(MessageListener el)	-
void close()	SessionCloseException
void abort()	-
void sendObject(Object object)	TLMException,SessionException
void send(byte[] buffer)	TLMException,SessionException
void send(byte[] buffer, int off, int len)	TLMException,SessionException
void sendEOM(byte[] buffer)	TLMException, SessionException
void sendEOM(byte[] buffer, int off, int len)	TLMException,SessionException
int sendObjectReqAck(Object object)	TLMException,SessionException
int sendEOMReqAck(byte[] buffer)	TLMException,SessionException
int sendEOMReqAck(byte[] buffer, int off, int len)	TLMException,SessionException
void sendEOM()	SessionException
int sendEOMReqAck()	SessionException
int blockingSendObject(Object object)	TLMException,SessionException
void blockingSend(byte[] buffer)	TLMException,SessionException
void blockingSend(byte[] buffer, int off, int len)	TLMException,SessionException
void blockingSendEOM(byte[] buffer)	TLMException,SessionException
void blockingSendEOM(byte[] buffer, int off, int len)	TLMException,SessionException
int blockingSendObjectReqAck(Object object)	TLMException,SessionException
int blockingSendEOMReqAck(byte[] buffer)	TLMException,SessionException
int blockingSendEOMReqAck(byte[] buffer, int off, int len)	TLMException, SessionException
void blockingSendEOM()	SessionException
int blockingSendEOMReqAck()	SessionException
DataMessage recv()	SessionException
DataMessage blockingRecv()	SessionException
void Session.confirm()	-
void Session.confirm(status)	-
Event getStatus()	-
Event blockingGetStatus()	-
String getStatus(int id)	-

by calling the `serialize` method of `ServerSession`. After creating the session, the peers can interact symmetrically, using `send`-like and `recv`-like methods or callbacks. FTSL provides a variety of `send` methods to allow the application to i) send the messages in small pieces for the sake of performance; ii) identify which level of acknowledgment (Application or Session Level) is expected for each particular message; and iii) to allow the application to send the messages in a blocking or non-blacking manner. Since the messages can be sent in pieces, the applications must mark the end of message (EOM). Some examples of the `send` methods are `sendEOM`, `sendEOMReqAck`, and `blockingSend`. The receive operation can also occur in a blocking or non-blocking flavors. The application calls the `confirm` method to notify the peer that it took care of a message, whenever such message requests for an application-level acknowledgment. Two methods, `getStatus` and `blockingGetStatus`, can be used by the application to get the status of the messages sent.

We suffix all the send methods with an `EOM`, whenever the method bounds the *end of message*, e.g., `sendEOM`. One should note that a single message may use a single method that should have the `EOM` suffix. To identify the acknowledgment level, FTSL offers some more `send` methods. The applications can ask for the receiver's confirmation using the methods whose names end by `ReqAck`, such as `sendEOMReqAck`. In addition to the above `send` methods, FTSL provides blocking methods that start with `blockingSend`.

All the `send` methods throw an `SessionException` when the session is already closed and a `TLMException` when the length of the message is larger than the total Sent Buffer's size.

The Peers can simply abort the session using the `abort` method, which closes the connection and the session immediately. To close the session safely, peers should use the `close` method. This prevents the application from sending new messages and waits until all messages reach the peer. If the connection is crashed then the FTSL throws `SessionCloseException`. In this case the application can use `abort` to close the session.

We made the source code of FTSL publicly available for download (URL: http://eden.dei.uc.pt/~naghmeh/ftsl/index.html). The source code includes three basic examples to demonstrate how to use the FTSL's API.

V. Evaluation of FTSL

In this section we evaluate the performance, overhead and scalability of FTSL, by comparing it to alternative solutions. We also verify FTSL's behavior in the presence of failures and migration operations. We implemented three versions of a Java client-server application, using TCP, RSocket, and FTSL. We used two computers sharing the same Local Area Network. We ran all the clients on a single process, using different threads on a Mac OS X, version 10.6.7, with a 2.4GHz Intel Core 2 Duo processor, 4GB of RAM and 3MB of cache. The server ran on a virtualized infrastructure with Linux version 2.6.34.8, with a 2.8 GHz Intel processor with four cores, 12 GB of RAM and 8 MB of cache. We consider three operations, `Invoke1`, `Invoke2` and `Invoke3`. They receive a small string and return another small string. The difference is that `Invoke1` replies immediately, `Invoke2` sleeps 1 millisecond (ms) before replying, whereas `Invoke3` sleeps 2 ms. The reason why we put the server threads to sleep is to minimize interference with our results. Despite this minimization, one should be aware that putting a thread to sleep and waking it up takes around 0.08 ms on the machine where we ran the server (and 0.15 on the client machine). To determine this number we ran a single-threaded program that slept for 1 ms 1000 times.

A. *FTSL's Performance*

Figure 3 shows the latency of the plain TCP, RSocket, and FTSL, for different numbers of clients. Latency is simply the round-trip-time of the request-response interaction. The plot also shows the performance degradation of FTSL compared to a plain TCP implementation on the right side vertical axis. Degradation is the difference between the latency of FTSL and TCP relative (divided by) FTSL's latency ($(FTSL - TCP)/FTSL$). Our initial results showed that RSocket is

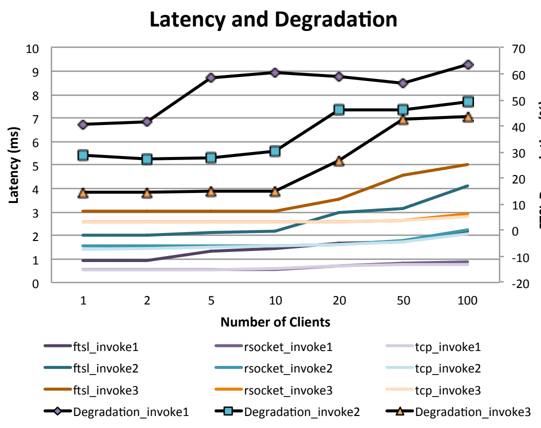

Fig. 3. Latency and Degradation of FTSL

particularly slow due to the Nagle's algorithm [24], which we could not switch off. To do a fair comparison, we have implemented that possibility in RSocket. The figure makes the processing time of the different `Invoke` operations apparent. Inevitably, TCP is the most efficient option (because both RSocket and FTSL use it underneath). With only 1 client TCP and RSocket take a little bit more than half the time of FTSL to call `Invoke1`. An important point here is that a significant part of the overhead we observe for a growing number of clients comes from putting threads to sleep and waking them back. First, we notice that 100 clients, take $0.08 * 100/4 = 2$ ms in the process. However, since they do not end up doing the actions completely in parallel (say client 1 can finish before client 100), the overhead of the 100 server threads is not exactly seen as 2 ms in each client. TCP and RSocekt have an overhead slightly over 0.6 ms (`Invoke2`), whereas the 100 clients take approximately 3 times more to run than a single one (because they run in the same dual core client). This is consistent with the latency growth we observe for `Invoke2` and `Invoke3`. `Invoke1` is faster because the server does not sleep before responding. In FTSL, the growth in latency is proportionally larger to the number of threads it has: as we will show in Figure 6, this is typically 3 times more than TCP and RSocket for 100 clients.

To examine the throughput, the client sends a large number of requests to the server (1000 in our tests). We set the server to send one reply to the client after receiving and processing all the requests, so that the client can calculate the overall time it takes from the first request it sends until the server finishes the last request (the impact of the extra reply in the throughput measurements is ignored). In Figure 4 we show the results obtained using FTSL, TCP and RSocket. The values shown in the figure for degradation are calculated using the throughput of TCP minus the throughput of FTSL divided by the throughput of TCP ($(TCP - FTSL)/TCP$).

In the left hand-side of Figure 4 we show a complete view, whereas in the right hand-side we show a partial view. The numbers shown represent the average throughput observed by each client. This means that the overall throughput at the server actually never decreases. What we can see is that each client may notice a sharp reduction in performance from its own point of view, when the server runs many more threads. The results show the FTSL's throughput for `Invoke2` and `Invoke3` is almost the same as TCP's throughput, except when 100 clients are running. This means that although FTSL's latency suffers from its extra operations and threads, the simultaneous reading from the socket compensates it and leads to the same throughput as FTSL. With 100 clients, the additional threads of FTSL, which serve to acknowledge messages and to read data from the socket, make it pay a reasonably small cost. For the `Invoke1` TCP and RSocket's throughput decreases when the number of the clients increases (due to the large number of the threads running). In the same case, FTSL shows much lower throughput than TCP and RSocket. This is caused by the FTSL's latency, which leads to an extra overhead in this scenario because the requests are continuously sent and FTSL needs to keep more messages in its own buffer. Nevertheless, future implementations of FTSL may consider the elimination of many of these threads (e.g., sharing them among different connections).

B. *FTSL's Scalability*

To evaluate the scalability of FTSL, we ran a multicast test scenario and compared it with UDP multicast and JMS, presumably the fastest, and a slow robust option, respectively. UDP is assumedly not reliable, as messages may be lost or delivered out of order. Unlike this, typical implementations of JMS use a powerful but heavy server node between publishers and subscribers, thus delaying communication. Communication cannot flow back to the publisher, unless it prepares destinations for reply messages. Relocation of the server does not seem reasonable to do, but one may trivially move or replace communicating peers. We do not include Java's Remote Method Invocation (RMI) in this comparison, because it sports a blocking request-response interaction that would be difficult to twist in our tests.

To evaluate the scalability of FTSL in comparison to the other solutions, we designed a test where several clients (from one to 500) connect to the server, before getting some data back. Since in this test, the FTSL client asks the server for the application layer confirmation, we ran two different scenarios with JMS as well, to make a fair comparison: one without confirmation and one with confirmation. Figure 5 shows the results. The scalability in these tests is defined as the number of clients served simultaneously in a unit of time. As expected, the UDP multicast is more scalable than FTSL (almost two times) and FTSL is more scalable than JMS (almost two times). If we need to confirm the JMS publisher about the reception of the messages, the scalability decreases almost to half. Overall, FTSL ensures the delivery of messages much faster than JMS.

C. *FTSL's Overhead*

To examine FTSL's overhead, we used Oracle's JConsole to monitor the heap memory, CPU usage, and the number of threads running on the server. We used varying numbers of clients, between 1 and 100. Each client sends 100 objects

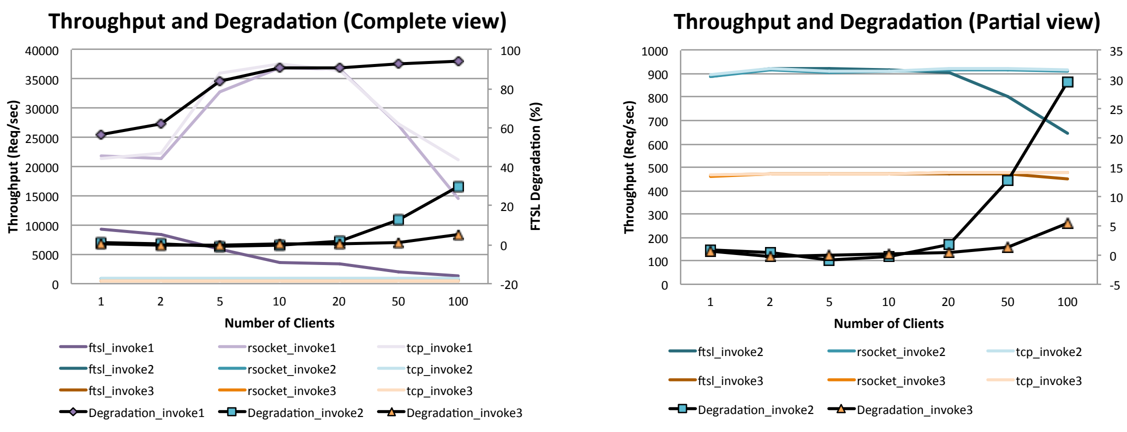

Fig. 4. Throughput and Degradation of FTSL

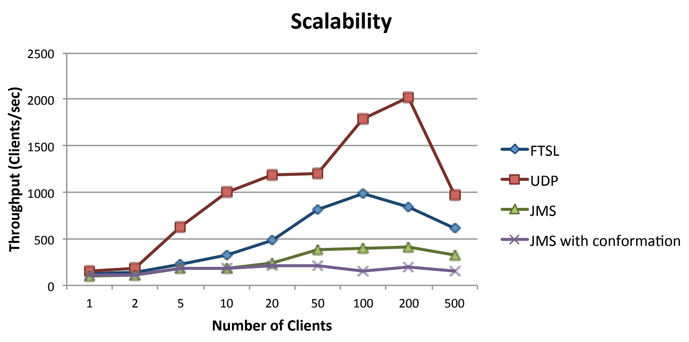

Fig. 5. Scalability of FTSL in comparison to JMS and UDP multicast

to the server per second during 5 minutes. The server writes each of these objects, carrying an integer and a name to the disk. Figure 6 compares the utilization of resources. As shown in the figure, TCP and RSocket use almost the same amount of resources. The extra complexity of FTSL imposes a little extra price in the CPU utilization of the server. Memory utilization of FTSL is higher than in TCP, most likely due to the extra buffering and the extra threads. We can see that each connection has two more threads in FTSL than TCP. One of these threads is for reading the messages from the TCP Socket and the other one is for acknowledgement.

D. Network Failure and Recovery

To evaluate FTSL's recovery from failures, we emulate connection crashes, by switching off the network interface, while the client sends a long file (e.g. music video) to the server. After one minute we resume the connection. For 100 repetitions of the test, we observed that, while the first connection to the server takes 15 ms in average, reconnection plus sending lost messages took an average of 26 ms, which, we believe, is quite fast.

E. Serialization and Migration

Figure 7 shows the average time of an FTSL server migration for 30 trials. DeserializationNS is the time the server

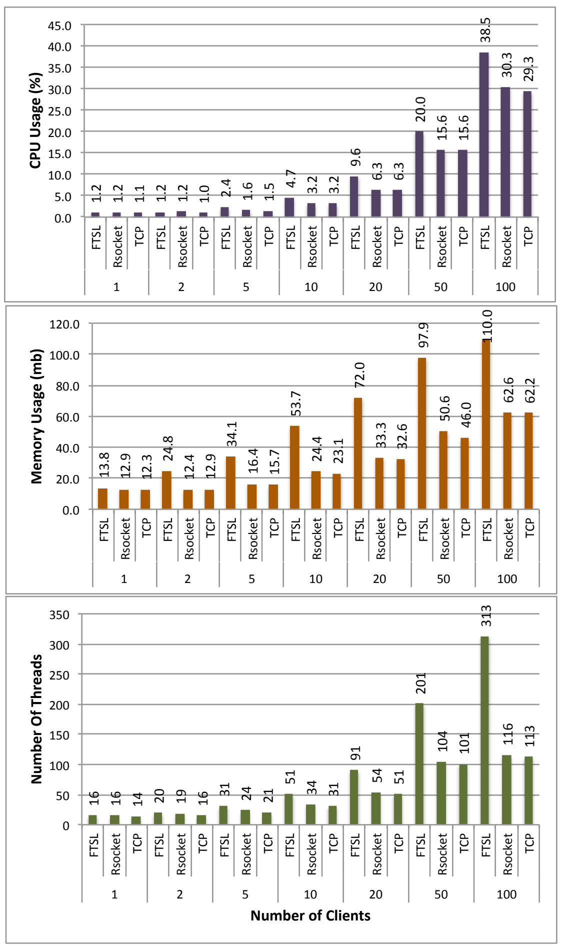

Fig. 6. Resource Usage

consumed communicating with the naming server during deserialization. Serialization is the time the server took to generate an object and save it to disk. Deserialization time refers to the reverse operation. Server Migration is the time it took to manually restart the server on a different port. ClientReconnection is the time it took for the last client to connect, once the server is ready. We varied the number of clients with active connections to the migrating server between 1 and

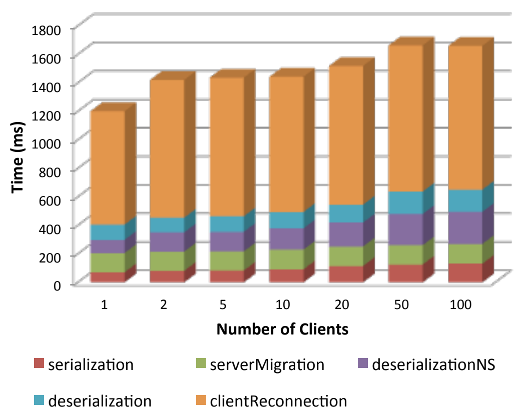

Fig. 7. FTSL Server Migration Time

100. We could conclude that server migration time is nearly independent on the number of clients. Despite this, the time taken for serialization and deserialization, and communication with the naming service during deserialization increases slowly with the number of clients (Figure 7). In addition, we observed that communication with the naming service, more precisely the rebind operation (deserializationNS), has an important impact on the overall migration time, while the reconnection time of the clients has the most significant impact. This time mainly depends on the frequency at which they check for the return of the server.

VI. Conclusions

In this paper we presented a Fault Tolerant Session Layer (FTSL), which covers several limitations of TCP for distributed applications that need fault tolerance and a higher level of reliability. With FTSL, such applications can 1) transparently recover from TCP connection crashes; 2) they can track the status of the messages that are sent, and 3) servers can be easily migrated to other machines. Our experiments show the feasibility of our approach. According to the results, performance and resource utilization of FTSL are comparable to the simpler RSocket. Even comparing to TCP, FTSL achieves reasonable results. By being reliable, by providing one-way communication and by facilitating migration for clouds, we believe that FTSL closes a gap that exists to this day in distributed systems. In the future, we plan to keep improving some aspects of FTSL, like the number of threads it uses that can cause a sensible reduction of performance in large servers.

Acknowledgments

This work was partially supported by the Portuguese Foundation for Science and Technology contract SFRH/BD/67131/2009 and by the project CMU-PT/RNQ/0015/2009, TRONE - Trustworthy and Resilient Operations in a Network Environment.

References

[1] Lorenzo Alvisi, Thomas C Bressoud, and Ayman El-Khashab. Wrapping Server-Side TCP to mask connection failures. *In proceeding IEEE INFOCOM*, 2001.

[2] David P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID*, pages 4–10, 2004.

[3] Devi Prasad Bhukya, Reeta Sony AL, and Gautam Muduganti. On web services based cloud interoperability. *International Journal of Computer Science Issues(IJCSI)*, 9(5), 2012.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.

[5] R. Braden. *RFC 1122 Requirements for Internet Hosts - Communication Layers*. Internet Engineering Task Force, October 1989.

[6] Ethan Cerami. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. O'Reilly Media, Inc., 2002.

[7] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[8] Troy Bryan Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1st edition, 1998.

[9] J. Hart. WebSphere MQ: connecting your applications without complex programming. *IBM WebSphere Software White Papers*, 2003.

[10] S. Horrell. Microsoft message queue. *Enterprise Middleware*, 1999.

[11] Information Sciences Institute. RFC 793, 1981. Edited by Jon Postel. Available at http://rfc.sunsite.dk/rfc/rfc793.html.

[12] JBoss Community. Community driven open-source middleware. http://www.jboss.org/. Last accessed on July, 4th 2013.

[13] Hal Jin, Jie Xu, Bin Cheng, Zhiyuan Shao, and Jianhui Yue. A fault-tolerant TCP scheme based on multi-images. In *IEEE Pacific Rim Conference on Communications Computers and Signal Processing (PACRIM 2003)*, pages 968–971, Victoria, Canada, 2003.

[14] Akane Koto, Hiroshi Yamada, Kei Ohmura, and Kenji Kono. Towards unobtrusive VM live migration for cloud computing platforms. In *Proceedings of the Asia-Pacific Workshop on Systems*, page 7, 2012.

[15] M. Litzkow and M. Livny. In *8th International Conference of Distributed Computing Systems,*.

[16] M. Lui, M. Gray, A. Chan, and J. Long. Scaling your spring integration application. *Pro Spring Integration*, pages 529–559, 2011.

[17] D. Maltz and P. Bhagwat. TCP splicing for application layer proxy performance. *research report 21139, IBM research division.*, 1998.

[18] M. Marwah and Sh. Mishra. TCP server fault tolerance using connection migration to a backup server. *In proceeding international conference on dependable systems and networks (DSN)*, pages 373–382, 2003.

[19] B. Nicolae and F. Cappello. Blobcr: Efficient checkpoint-restart for hpc applications on iaas clouds using virtual disk image snapshots. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1 –12, nov. 2011.

[20] M. Orgiyan and C. Fetzer. Tapping tcp streams. In *Network Computing and Applications, 2001. NCA 2001. IEEE International Symposium on*, pages 278–289, 2001.

[21] C. Metz R. Stewart. Sctp: new transport protocol for tcp/ip. *IEEE Internet Computing*, Vol. 5, No. 6:pp. 64–69, 2001.

[22] Ajith Ranabahu, E. Michael Maximilien, Amit Sheth, and Krishnaprasad Thirunarayan. Application portability in cloud computing: An abstraction driven perspective. *IEEE Transactions on Services Computing*, 99(1):1, 5555.

[23] R.Ekwall, P.Urban, and A.Schiper. Robust TCP connections for fault tolerant computing. *In proceeding 9th international confeence on parallel and distributed systems (ICPADS)*, 19:501—508, 2002.

[24] Jin Rencheng, Meng Xiao, Meng Lisha, and Wang Liding. A design of efficient transport layer protocol for wireless sensor network gateway. In *2nd International Conference on Signal Processing Systems*, pages V1–775 –V1–780, July 2010.

[25] M. Richards, R. Monson-Haefel, and D. A. Chappell. *Java message service*. O'Reilly Media, 2009.

[26] Kassem Saleh, Hasan Ural, and Anjali Agarwal. Modified distributed snapshots algorithm for protocol stabilization. *Computer Communications*, 17(12):863–870, December 1994.

[27] Zhiyuan Shao, Hai Jin, Bin Cheng, and Wenbin Jiang. ER-TCP: an efficient fault-tolerance scheme for cluster computing. *The Journal of Supercomputing*, 2007.

[28] Gurudatt Shenoy and Suresh K Satapati. HYDRANET-FT: network support for dependable services. *In international conference on distributed computing systems*, 2000.

[29] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 95–106, 2002.

[30] Yi Zhao and Wenlong Huang. Adaptive distributed load balancing algorithm based on live migration of virtual machines in cloud. In *Fifth International Joint Conference on INC, IMS and IDC, 2009. NCM '09*, pages 170–175, 2009.